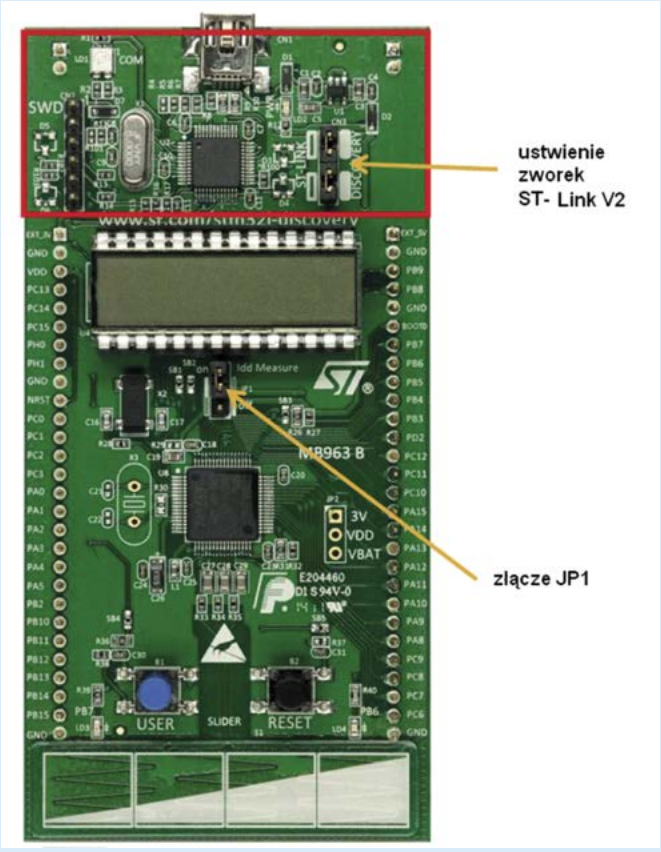
STM32 – tryby obniżonego poboru mocy (3)

Do tej pory poznaliśmy budowę zestawu ewaluacyjnego STM32L Discovery, metody pomiaru natężenia bardzo małego prądu oraz działanie i programowe wprowadzenie trybów ograniczonego poboru energii. Teraz przyszła pora na praktyczne rozważania, w jaki sposób można optymalnie i racjonalnie gospodarować zapotrzebowaniem na energię w ramach dostępnych trybów jej oszczędzania.

Dla potrzeb nauki zastosujemy już nam znany zestaw ewaluacyjny STM32L Discovery z wbudowanym programatorem/debugerem ST-Link V2. W programach testowych dostarczanych z modułem prąd jest mierzony przez wbudowany układ i wyświetlany na LCD. To wygodne rozwiązanie, ale wymaga napisania sporego programu wykonującego pomiary. Jego praca może w naszych przykładach powodować zafałszowanie wyników i przez to wyciąganie nieprawidłowych wniosków. Na szczęście producent modułu przewidział możliwość pomiaru prądu za pomocą zewnętrznego mikroamperomierza. Układ pomiaru prądu można włączać i wyłączać zworką JP1 (**rysunek 1**). Kiedy jest ona w położeniu ON, to mikrokontroler jest zasilany przez układ pomiaru prądu. Położenie zworki w pozycji OFF powoduje ominięcie układu pomiarowego. Po całkowitym usunięciu zworki, pobór prądu można mierzyć amperomierzem włączonym pomiędzy piny 1 i 2 złącza JP1. Dokładny pomiar małych prądów może wymagać precyzyjnego mikroamperomierza, spełnienia określonych warunków pomiaru, tym bardziej, że prąd pobierany przez mikrokontroler ma charakter impulsowy. Nam jednak nie będzie chodziło o dokładność pomiaru, ale o zaobserwowanie tendencji zmian poboru prądu. Dlatego do pomiaru zastosowałem popularny multimetr Metex M-3800 na zakresie pomiarowym prądu stałego 200 µA. Jak już wspomniałem, zaciski miernika mają być włączone pomiędzy piny 1 i 2 złącza JP1.



Rysunek 1. Konfiguracja modułu STM32L Discovery

Wpływ konfiguracji kompilatora

Pisząc program za pomocą kompilatora języka wysokopoziomowego (w przypadku mikrokontrolerów będzie to zazwyczaj kompilator C lub C++) możemy w pewnym stopniu decydować o właściwościach kodu programu wynikowego. Zdecydowana większość kompilatorów ma możliwość ustawiania stopnia i sposobu optymalizacji kodu wynikowego programu. Może on być optymalizowany pod względem szybkości wykonywania lub wielkości kodu wynikowego. Często w trakcie pisania i debugowania doświadczeni programiści wyłączają optymalizację. Pomaga to głownie w debugowaniu. Dopiero działający, potencjalnie bezbłędny program jest poddawany optymalizacji.

Można sobie zadawać pytanie: jak konfiguracja kompilatora może mieć wpływ na ilość pobieranej energii? Intuicyjnie czujemy, że może to mieć związek z szybkością wykonywania programu. W pierwszej części artykułu napisałem, że pobór prądu przez układy cyfrowe taktowanie zegarem jest tym większy, im większa jest częstotliwość taktowania. Ta zależność sprawdza się dla mikrokontrolera taktowanego ze stałą częstotliwością i pracującego z wykluczeniem trybów oszczędzania energii. W uproszczeniu można przyjąć, że pobór mocy przez mikrokontroler jest stały w czasie, tak jak to zostało pokazane na **rysunku 2**.

Częstotliwość taktowania można dobrać na tyle małą, aby była wystarczająca do wykonania zadań stawianych

Rysunek 2. Pobór mocy przez mikrokontroler taktowany ze stała częstotliwością

przed aplikacją. Pomijając problem określenia tej częstotliwości, to nie jest to rozwiązanie optymalne z punktu widzenia poboru energii. W wielu aplikacjach mikrokontroler nie jest stale obciążony wykonywanymi zadaniami. Kiedy nic nie ma do zrobienia, to nadal pobiera tę samą ilość energii, jak w trakcie wykonywania obliczeń czy algorytmów sterowania. Dlatego są załączane tryby oszczędzania energii, kiedy CPU nic nie ma do roboty lub kiedy trzeba wykonać coś, co nie wymaga dużych prędkości działania. Ograniczenie poboru energii w tych trybach zależy od rozwiązań zastosowanych przez projektantów rdzenia i od producentów konkretnego mikrokontrolera (rozwiązania układów peryferyjnych, technologii produkcji itp.). Spróbujmy się zastanowić czy można dodatkowo ograniczyć pobieranie energii w czasie, gdy mikrokontroler pracuje z wyłączonymi trybami oszczędzania, czyli kiedy jest taktowany z najwyższą częstotliwością roboczą i zasilany napięciem nominalnym.

Intuicyjnie wydaje się, że tu również należy ograniczać pobór prądu przez zmniejszenie częstotliwości taktowania do jak najniższej wartości. Załóżmy, że mikrokontroler jest w stanie niskiego poboru energii, z którego jest wybudzany sekwencyjnie po to, aby wykonać obliczenia lub sekwencję sterującą. Po wykonaniu tego zadania jest natychmiast ponownie usypiany. Okazuje się, że w takich wypadkach jest najlepiej, aby zadanie wykonywane po wybudzeniu było realizowane jak najszybciej. Szybkość pracy jest zależna nie tylko od częstotliwości taktowania, ale również od optymalizacji wykonywanego kodu. Przy tej samej prędkości taktowania mikrokontrolera kod wynikowy zoptymalizowany pod kątem jak największej szybkości wykonywania powinien wykonać się szybciej, niż kod nieoptymalizowany. Wtedy również ilość pobranej energii będzie mniejsza.

Przy stałym napięciu zasilania i stałym prądzie pobieranym przez mikrokontroler moc pobierana jest wyliczana z zależności $P=V\times I$, gdzie: V – napięcie zasilania, I średni (całkowany) prąd pobierany przez mikrokontroler). Pobierana energia jest wyliczana z zależności $E=P\times t$, gdzie: P – moc pobierana, t – czas. Łatwo domyślić się, że im krótszy czas, tym mniejsza energia pobierana.

Wróćmy do naszego przykładu z sekwencyjnym wybudzeniem. Ponieważ nie mamy możliwości bezpośredniego mierzenia poboru energii, to będziemy mierzyli prąd pobierany przez mikrokontroler, bo jego wartość przy stałym napięciu jest proporcjonalna do pobieranej mocy i energii. Wartość prądu pobieranego przez układ zmienia się impulsowo. Mikroamperomierz prądu stałego pokaże wartość uśrednioną i dla naszych celów będzie to wystarczające.

Program

Do testowania wpływu optymalizacji kodu przez kompilator zastosujemy program testowy, który:

- Włącza i konfiguruje niezbędne układy peryferyjne, a wyłącza niepotrzebne.
- Konfiguruje timer wybudzający mikrokontroler co 40 ms.
- Wprowadza tryb STOP oszczędzania energii.
- Po każdym wybudzeniu oblicza prosta sumę kontrolną z elementów tablicy umieszczonej w pamięci SRAM oraz średnią geometryczną z trzech zmiennoprzecinkowych argumentów.

Konfiguracja systemu i pętla główna programu został pokazana na listingu 1

Listing 1. Konfigurowanie układów peryferyjnych i obliczenia wykonywane po wybudzeniu

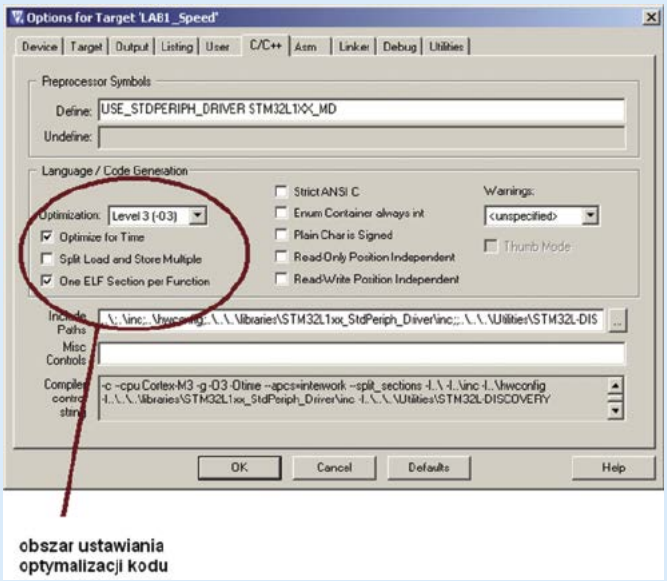
```
static volatile char ARRAY[64] = {0xAA,0x22,0x17,0xB1,0x5
5,0x15,0x23,0x75,
0xF0,0x41,0x19,0x1A,0x81,0x99,0x10,0x51,
0x10,0x11,0x51,0x13,0x14,0x01,0x98,0x88,
0x18,0x91,0x07,0x01,0x22,0x61,0x24,0x51,
0x26,0x05,0x36,0x99,0x30,0x21,0x32,0x00,
0x15,0x16,0x67,0xBB,0xFF,0x50,0x13,0x91,
0x71,0x42,0x61,0x11,0x63,0x91,0x10,0x10,
0x94,0xFF,0x17,0x18,0x29,0x71,0x31,0x55};

static volatile float f_coefA = 0.121f;
static volatile float f_coefB = 0.785f;
static volatile float f_coefC = 0.349f;
int main(void)
{
  unsigned int index;
  unsigned char result;
  volatile float f_root;
/* konfiguracja układów peryferyjnych  */
  SYSTEM_Config();
/*konfiguracja zegara, trybu osczedzania energii
I licznika wybudzeń*/
  LAB1_GetReady();
/* pętla nieskończona  */
  while(1)
  {
/* sygnalizacja wykonywania obliczeń CRCpo wybudzeniu  */
    EVENT_HIGH();
/*wartości początkowe zmiennych
    index = 64;
    result = 0;
    f_root = 0.0f;
/* wyliczenie sumy CRC */
    while(index != 0)
    {
      result ^= ARRAY[--index];
    }
/* Sprawdzenie wyniku  */
    if (result != CHECKSUM_RESULT)
    {
      break;//koniec liczenia CRC
    }
    EVENT_LOW();
/* Sygnalizacja obliczeń średniej */
    EVENT_HIGH();
/* Mnożenie współczynników  */
    f_root = f_coefA * f_coefB * f_coefC;
/* obliczanie pierwiastka */
    f_root = cbrtf(f_root);
    EVENT_LOW();
    __wfi();//wejście w tryb STOP
  }
}
```

Procedura *SYSTEM_Config()* konfiguruje zarządzaniem pamięcią Flash, blokami RCC, PWR oraz portami GPIO, układem przerwań i zegarkiem RTC. Jak już wiemy, również tryby oszczędzania energii wymagają skonfigurowania przed ich wprowadzeniem rozkazem *wfi*. Tę czynność wykonuje procedura *LAB1_GetReady()* pokazana na **listingu 2**.

Poziom optymalizacji kodu wynikowego w pakiecie uVision 4 jest ustawiany w oknie *Project -> Options for*

Listing 2. Konfigurowanie trybu STOP i okresu wybudzania

```
void LAB1_GetReady(void)
{
/* właczenie zegara dla modułu PWR */
  *(__IO uint32_t *) RCC_APB1ENR_PWREN_BB = (uint32_t)
SET;
  /* Wprowadzenie trybu STOP jako  deepsleep */
  *(__IO uint32_t *) PWR_CR_PDDS_BB = (uint32_t)RESET;
/* Wewnętrzny regulator napięcia ustawiony w tryb Low
Power Mode w czasie trybu STOP */
  *(__IO uint32_t *) PWR_CR_LPSDSR_BB = (uint32_t)SET;
/* ustawienie bitu SLEEPDEEP rejestru  Cortex System
Control Register */
  SCB->SCR |= SCB_SCR_SLEEPDEEP;
/* ustawienie czasu cyklicznego wybudzania  w ms */
  RTC_SetWakeUp(40);
/* zablokowanie protekcji zapisu rejestrów zegara */
  RTC->WPR = 0xCA;
  RTC->WPR = 0x53;
/* zablokowanie zdarzenia EXTI dla RTC Wake-up */
  EXTI->EMR &= ~EXTI_Line20;
/* odblokowanie przerwania EXTI dla RTC Wake-Up */
  EXTI->IMR |= EXTI_Line20;
/* zerowanie flagi  RTC wake-up  */
  *(__IO uint32_t *) RTC_ISR_WUTF_BB = (uint32_t)RESET;
/* odblokowanie timera Wakeup Timer*/
  *(__IO uint32_t *) RTC_CR_WUTE_BB = (uint32_t)SET;
}
```



Rysunek 3. Ustawianie optymalizacji w pakiecie uVision

Tabela 1. Wyniki pomiaru prądu w zależności od stopnia optymalizacji programu

Optymalizacja kodu	Zmierzony prąd [µA]
Brak	33,5
Ustawienia domyślne	30,2
Pod względem prędkości	22,9

Target w zakładce C/C++ i obszarze *Language/Code Generation* (**rysunek 3**). Dla celu testów program skompilowano trzykrotnie: bez optymalizacji, z ustawieniami domyślnymi i z optymalizacją prędkości działania (*Optimize for Time*). Przy każdej z prób prąd był mierzony multimetrem na zakresie 200 µA. Wyniki testu zostały umieszczono w **tabeli 1**.

Wyniki wyraźnie różnią się między sobą i najmniejszy pobór prądu jest po skompilowaniu programu z kryterium najwyższej prędkości działania (najkrótszego czasu). Trzeba jednak pamiętać o dwóch rzeczach. Po pierwsze, w tej aplikacji mikrokontroler jest przez większość czasu uśpiony. Jest wybudzany sekwencyjnie po to, aby wykonać obliczenia i po realizacji tego zadania zostaje natychmiast uśpiony ponownie. Jest to często spotykane działanie, ale w innych aplikacjach wynik porównania nie musi być tak spektakularny. Po drugie, wykonywane obliczenia tak dobrano, aby optymalizacja kodu wyraźnie przyspieszyła ich wykonywanie. Mimo tego ten przykład wyraźnie pokazuje pewną tendencję znaną konstruktorom zajmującym się układami Low Power. W wielu aplikacjach warto usypiać mikrokontroler tak często i „głęboko" jak się da, ale jak już jest wybudzony, to powinien pracować tak szybko, jak to tylko możliwe. Wykona wtedy on swoje zadanie szybko i w końcowym efekcie pobierany prąd będzie mniejszy, niż w przypadku, gdy po wybudzeniu będzie pracował wolniej.

Architektura rdzenia, a pobór energii

Mikrokontrolery z rdzeniami ARM Cortex-M mogą znacznie różnić się. Nie chodzi tu o naturalne różnice wynikające z wyposażenia w układy peryferyjne, pamięć programu Flash czy pamięć danych SRAM, ale o budowę samego rdzenia. Rdzenie różnią się architekturą i listą rozkazów. W **tabeli 2** umieszczono zestawienie właściwości poszczególnych rdzeni ARM Cortex-M.

Rdzenie Cortex M0, M0+ i M1 wykonują większość instrukcji z 16-bitowych rozkazów Thumb oraz niektóre instrukcje zestawu Thumb2. Z zestawu 32-bitowych instrukcji Cortex-M są wykonywane tylko instrukcje sprzętowego dzielenia w jednym lub 32 cyklach (zależnie od implementacji w krzemie). Rdzeń Corte-M3 wykonuje wszystkie 16-bitowe rozkazy Thumb i Thumb2 oraz wszystkie 32-bitowe rozkazy ARM Cortex-M. Rdzeń Cortex-M4 jest dodatkowo wyposażony w rozkazy jednostki DSP.

Oprócz zmiennej listy rozkazów rdzenie różnią się architekturą. Przyjęło się przekonanie, że szybkie mikrokontrolery ARM mają architekturę typu Harvard, czyli z rozdzielonymi magistralami danych i programu. Można dzięki temu w pojedynczych cyklach zegarowych uzyskać dostęp do kodu rozkazu z pamięci Flash i argumentu z pamięci SRAM. Tak jest w rdzeniach Cortex-M3 i Cortex-M4. Proste i w założeniu tanie rdzenie Cortex-M0, -M0+ i -M1 maja architekturę Von Neumanna, co może być zaskoczeniem dla wielu użytkowników mikrokontrolerów z rdzeniem ARM Cortex. Architektura Von Neumanna ma wspólną magistralę danych i programów i z tego powodu wykonywanie rozkazów jest wolniejsze.

Mikrokontroler STM32L152RBT6 użyty w module STM32L Disovery ma rdzeń Cortex M3, a wiec jego rdzeń (ARM V7-M) ma architekturę typu Harvard i obsługuje wszystkie rozkazy z listy Thumb, Thumb2 i ARM Cortex-M. Istnieje możliwość takiego skompilowania programu w języku C, by kod wynikowy składał się tylko z 16-bitowych rozkazów Thumb i Thumb2 lub wykorzystywał także rozkazy 32-bitowe. Możemy spróbować sprawdzić, jak dostępny zbiór rozkazów wpływa na pobór energii przez mikrokontroler. Możemy się teraz domyślać, że stosowanie 32-bitowych rozkazów spowoduje szybsze wykonywanie programu, a co za tym idzie mniejszy pobór prądu. Ustawień dokonuje się w oknie konfiguracji projektu *Project -> Options for Target* w zakładce *Device*. Dla kompilacji z rozkazami 16 bitowymi z listy Thumb I Thumb2 wybieramy Device Cortex M0, a dla kompilacji z rozkazami 32 bitowymi Device Cortex M3. – rysunek 4

Testowany program jest dokładnie taki sam jak w poprzednim przykładzie. Optymalizacja kompilacji jest ustawiona na najszybsze wykonywanie programu

Wyniki pomiarów umieszczono w **tabeli 3**.

Po analizie pierwszego przykładu taki wynik nie jest zaskoczeniem. Możliwość użycia 32 bitowych rozkazów

Tabela 2. Wybrane właściwości rdzeni ARM Cortex-M

Rdzeń	Thumb	Thumb2	Dzielenie sprzętowe	Jednostka DSP	Architektura ARM	Architektura Rdzenia
Cortex-M0	Większość	Podzbiór	1 lub 32 cykle	Nie	ARMv6-M	Von Neumann
Cortex -M0+	Większość	Podzbiór	1 lub 32 cykle	Nie	ARMv6-M	Von Neumann
Cortex -M1	Większość	Podzbiór	3 lub 33 cykle	Nie	ARMv6-M	Von Neumann
Cortex -M3	Wszystkie	Wszystkie	1 cykl	Nie	ARMv7-M	Harvard
Cortex-M4	Wszystkie	Wszystkie	1 cykl	Tak	ARMv7E-M	Harvard

Tabela 3. Pobór prądu w zależności od użytej listy rozkazów	
Lista rozkazów	**Zmierzony prąd [µA]**
16 Bitowe Thumb i Thumb2 (Cortex M-0)	30,3
Ustawienia domyślne	23,2

do obliczeń zmiennoprzecinkowych spowodowała szybsze ich wykonanie i w rezultacie wyraźnie mniejszy pobór prądu.

Optymalizacja kodu niezależna od kompilatora

Już wiemy, że program, który wykonuje się szybciej wymaga mniejszej ilości energii zasilającej. Optymalizacja wbudowana w kompilator to nie jest jedyna możliwość przyspieszenia działania. Można próbować tak napisać program, by nawet po optymalizacji wykonywał się jeszcze szybciej. Żeby to robić trzeba próbować różnych konstrukcji programowych i analizować jak poradzi sobie z nimi kompilator. Do tego jest potrzebna znajomość asemblera i listy rozkazów. Tutaj pokażemy wpływ deklaracji długości zmiennej indeksującej wykonywanie pętli oraz zastąpienie pętli *while* ciągiem wywoływanych instrukcji. Do celów testowych zastosujemy procedurę liczącą sumą kontrolną crc po wybudzeniu mikrokontrolera ze stanu STOP sekwencyjnie, co 40 ms.

Na **listingu 3** pokazano procedurę liczenia crc wykonywaną w pętli *while*. Zmienna *index* licząca wykonywane obliczenia jest typu unsigned short int (16 bitów).

Po skompilowaniu, zapisaniu do pamięci Flash i uruchomieniu programu mikrokontroler pobiera ok. 13,2 µA. Teraz zmieniamy definicję zmiennej *index* na *unsigned int*, czyli typ 32-bitowy. Ta zmiana powoduje, że mikrokontroler pobiera ok. 10,8 µA. Jak widać, jest wyraźnie zauważalna różnica na korzyść zmiennej 32-bitowej. Zobaczmy jak to wygląda w asemblerze. Na **listingu 4** pokazano skompilowany kod sprawdzania warunku końca pętli dla zmiennej *index* typu unsigned short (16 bitów), a na **listingu 5** sprawdzania warunku końca dla zmiennej 32-bitowej.

Jeżeli pętla w programie nie wykonuje się wiele razy, a szczególnie zależy nam na ograniczeniu pobieranej energii, to warto rozważyć zastąpienie wykonywania instrukcji z pętli kolejnymi instrukcjami umieszczonymi w kodzie. Nie będzie wtedy potrzebne sprawdzanie warunku końca pętli i przez to program może się wykonywać szybciej. W naszym przykładzie zastąpimy pętlę 64-krotnym wywołaniem

```
temp ^= ARRAY[index];
index--;
```

Po takiej modyfikacji programu mikrokontroler pobiera ok. 6,5 µA prądu. Końcowe wyniki pomiarów umieszczono w **tabeli 4**.

Wpływ konfiguracji portów GPIO na pobór energii

W układach mikrokontrolerowych, w których pobór energii jest istotnym parametrem, trzeba zwracać uwagę na układy peryferyjne. Programista powinien konfigurować i włączać tylko te, które są niezbędne do działania aplikacji. Pozostałe muszą być wyłączone i nie powinny być taktowane. Jednak w każdym mikrokontrolerze jest jeden układ peryferyjny, którego całkowicie wyłączyć nie

Listing 3. Wyliczenie CRC

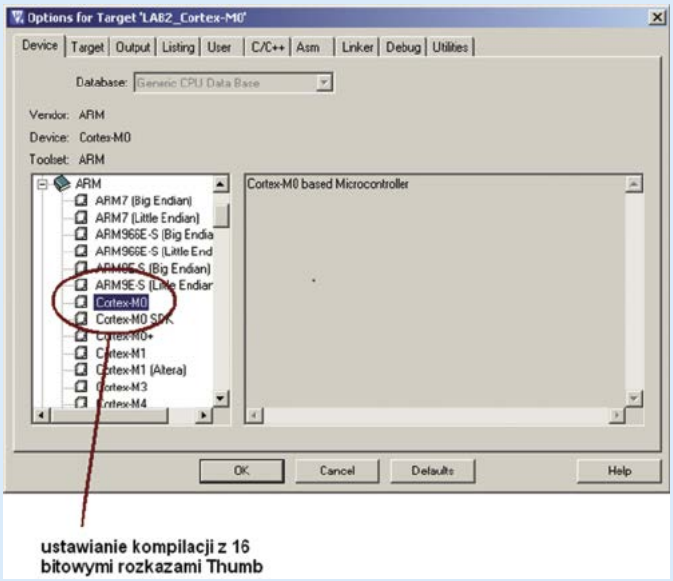
```
unsigned short int index;
unsigned char temp;
index = 64;
temp = 0;
/* wyliczenie sumy kontrolnej */
while(index != 0)
{
  temp ^= ARRAY[--index];
}
/* sprawdzenie wyniku  */
if (temp != CHECKSUM_RESULT) break;
```

Listing 4 sprawdzania warunku końca pętli dla 16 bitowej zmiennej index

```
73:                     index = 64;
0x08000832 2040         MOVS      r0,#0x40
...
...
0x08000836 1E40         SUBS      r0,r0,#1
0x08000838 0400         LSLS      r0,r0,#16
0x0800083A 0C00         LSRS      r0,r0,#16
```

Listing 5 skompilowany kod dla 32 bitowej zmiennej index

```
73:                     index = 64;
0x08000832 2040         MOVS      r0,#0x40
...
...
0x08000840 1E80         SUBS      r0,r0,#2
```



Rysunek 4. Wybranie rozkazów 16-bitowych

Tabela 4. Wyniki pomiaru dla różnych optymalizacji programowych	
Programowa optymalizacja kodu	**Zmierzony prąd [µA]**
Zmienna Index 16 bitowa	13,2
Zmienna Index 32 bitowa	10,8
Z „rozpisaną" pętlą	6,5

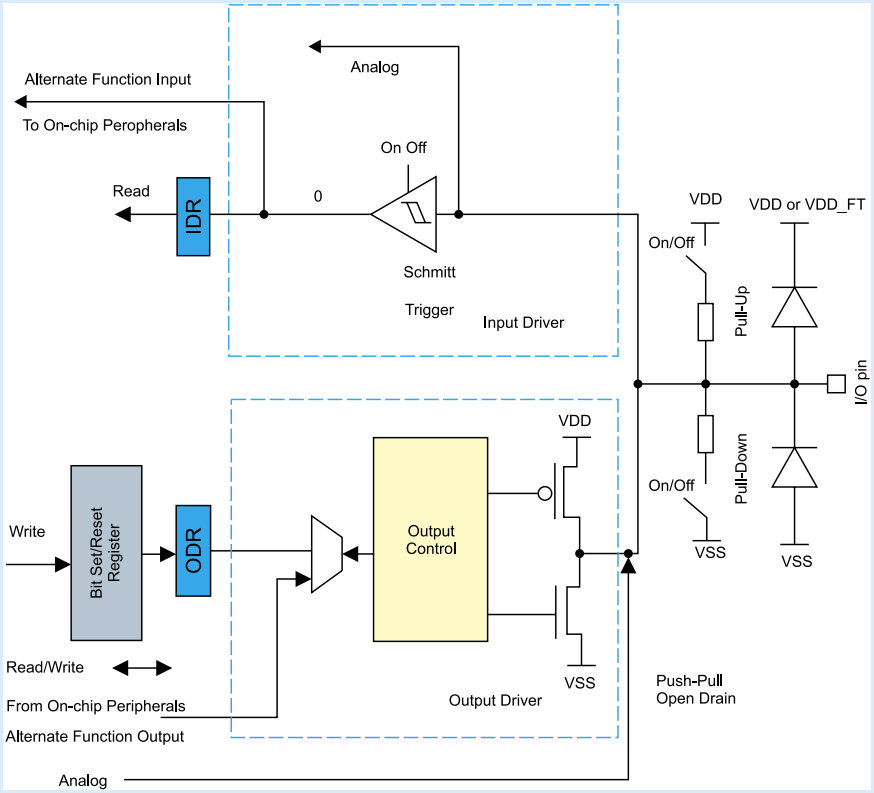
można. Są to uniwersalne porty GPIO. Schemat blokowy jednej linii portu pokazano na **rysunku 5**. Może one pracować w następujących trybach: wejściowa (*floating*), wyjściowa z poziomem niskim, wyjściowa z poziomem wysokim, wejściowa cyfrowa, wyjściowa analogowa. Dodatkowo, można włączyć rezystory podciągające do plusa zasilana (pull-up) lub do masy (pull-down). Podciąganie można włączyć nawet w konfiguracji portu wyjściowego.

Przy wielu możliwościach konfiguracji linii portów trzeba się zastanowić, w jaki tryb wprowadzić nieużywane porty I/O, aby pobór prądu był jak najmniejszy. Ponieważ mamy do dyspozycji moduł z możliwością pomiaru prądu, to optymalną konfigurację portów można ustalić eksperymentalnie. Do tego celu użyjemy programu, który

między innymi konfiguruje linie portów i wchodzi w stan STOP.

Na początek zobaczymy, jaki będzie pobór prądu, kiedy linie są konfigurowane przez procedurę *GPIO_Config* pokazaną na **listingu 6**. W mikrokontrolerach STM32 najmniejszy pobór prądu i największą odporność na zakłócenia uzyskuje się po ustawieniu wszystkich niewykorzystanych linii, jako wejścia analogowe. Poza tym muszą być wyłączone wszystkie rezystory podciągające.

Przy tak skonfigurowanych wejściach moduł pobiera poniżej 1 µA. Mój miernik pokazał ok. 0,5 µA, ale ze względu na zakres pomiarowy 200 µA, tę wartość trzeba traktować jako orientacyjną. Wybierzmy teraz jedną linie portu (PD2) i skonfigurujmy ją jako wejściową (float) bez rezystora podciągającego (**listing 7**).



Rysunek 5. Schemat blokowy linii portów GPIO STM32

Listing 6 konfiguracja linii portów jako analogowe wejścia.

```
void GPIO_Config(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
/* GPIO periph clocks enable */
  RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOA |
                         RCC_AHBPeriph_GPIOB |
                         RCC_AHBPeriph_GPIOC |
                         RCC_AHBPeriph_GPIOD |
                         RCC_AHBPeriph_GPIOE |
                         RCC_AHBPeriph_GPIOH, ENABLE );
/* ustawienie wszsytkich nie uzywanych linii jako wejścia
analogowe */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_400KHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //wejścia
analogowe
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //
wyłaczenie podciągania.
  GPIO_Init(GPIOC, &GPIO_InitStructure);
  GPIO_Init(GPIOD, &GPIO_InitStructure);
  GPIO_Init(GPIOE, &GPIO_InitStructure);
  GPIO_Init(GPIOH, &GPIO_InitStructure);
}
```

Listing 7 Konfiguracja linii PD2 jako wejściowej float

```
RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOD, ENABLE );
/* Konfiguracja PD2 jako linii wejściowej bez podciagania
*/
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Listing 8. Ustawienie linii PD2 w stan niski z podciąganiem Pull up

```
RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOD, ENABLE );
/* konfiguracja PD2 jako wyjsciowa z podciaganiem  Pull-
UP */
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOD, &GPIO_InitStructure);
/* ustawenie PD2 w stan niski  */
GPIO_ResetBits(GPIOD, GPIO_Pin_2);
```

Prąd pobierany wzrasta do ok. 7,5 µA i to tylko dla jednej linii zdefiniowanej jako wejściowa. Ustawienie linii jako wyjściowej wymaga szczególnej uwagi. W mikrokontrolerach STM32 włączenie podciągania jest niezależne od kierunku przepływu danych przez linię portu GIPO. W kolejnym przykładzie zdefiniujemy linię PD2 jako wyjściową, wyzerujemy ją, a jednocześnie włączymy podciąganie do plusa zasilania.

W tym przypadku pobór prądu wzrasta do ok. 85 µA. Jeżeli przydarzy się nam taki błąd, to może zniweczyć większość wysiłków wykonywanych w celu zmniejszenia poboru przez mikrokontroler.

Użycie układu DMA

Kanały DMA to szybki i wygodny mechanizm przesyłania danych z układów peryferyjnych bezpośrednio do pamięci bez użycia jednostki centralnej (CPU). Zbadamy teraz czy użycie DMA wpłynie na pobór prądu przez mikrokontroler. Do tego celu zostanie użyty program, który:

- Wprowadza tryb STOP.
- Jest wybudzany z tego trybu co 10 ms przez przerwania od timera.
- Za pomocą modułu A/C odczytuje 16 kolejnych próbek napięcia z wejścia analogowego PA0.
- Oblicza medianę próbek.

Tabela 5. Wyniki pomiarów dla różnych konfiguracji linii portów

Konfiguracja linii portów	Zmierzony prąd [µA]
Wszystkie nie używane linie jako wejścia analogowe	0,5
PD2 jako wejście float bez podciągania	7,5
PD2 jako wyjście w stanie nikim z włączonym pull-up	85

- Wyświetla wynik na wyświetlaczu LCD modułu STM32L Discovery.

W pierwszej implementacji próbki będą odczytywane z przetwornika A/C i przesyłane do bufora w pamięci RAM przez mikrokontroler (CPU). Po skonfigurowaniu przetwornika w pętli głównej jest wywoływana funkcja *acquireData()*, która odczytuje 16 kolejnych pomiarów z przetwornika (**listing 9**).

Pobór prądu wynosi teraz ok. 570 µA. Teraz konfigurujemy DMA do bezpośredniego przesyłania próbek z przetwornika A/C do bufora w pamięci RAM (**listing 10**).

Procedura acquireDataDMA konfiguruje i odblokowuje przetwornika ADC1, oraz kanał DMA1, zeruje licznik danych przesyłanych, przypisuje kanał DMA1 do przetwornika ADC1 i programowo uruchamia konwersję analogowo cyfrową (**listing 11**).

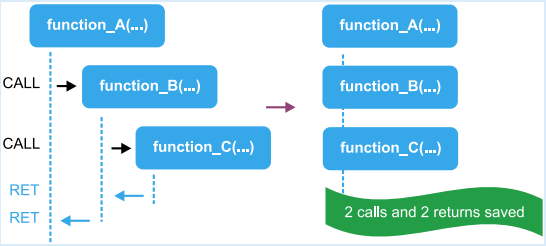
Tak skonfigurowany kanał DMA odczytuje zadana ilość danych z przetwornika A/C i przesyła bezpośrednio do pamięci RAM. Po przesłaniu wszystkich danych jest zgłaszane przerwanie, które wybudza mikrokontroler ze stanu STOP. Potem jest wyliczana i wyświetlana mediana z próbek. Pobór prądu teraz wynosi ok. 375 µA. Jak widać w tym przypadku pobór prądu spada dość wyraźnie, a bezpośrednie przesyłanie danych z układu peryferyjnego do pamięci przez kanał DMA nie jest wydumanym przykładem, ale często spotykaną sytuacją w algorytmach sterowania.

Podsumowanie

Programista tworzący program dla aplikacji, w której pobór energii ma kluczowe znaczenie powinien brać pod uwagę – oprócz możliwości wykorzystywania trybów oszczędzana energii – również zachowanie się mikrokontrolera w czasie, gdy pracuje przy nominalnym napięciu zasilającym i z nominalną częstotliwością taktowania.

Optymalizacja kodu jest jedną z ważnych metod ograniczania poboru energii. Usuniecie każdej niepotrzebnie wykonywanej instrukcji może spowodować wymierne korzyści. Ktoś oszacował, że w kodzie składającym się z 800 instrukcji każda pojedyncza instrukcja może się wykonywać 158×10^6 razy w ciągu 5 lat w aplikacji z mikrokontrolerem STM32 taktowanym częstotliwością 16 MHz. Jej usuniecie z kodu spowoduje, że procesor może być w stanie uśpienia przez 2,3 dnia dłużej w ciągu całego szacunkowego okresu użytkowania równego 5 lat. To daje już bardzo wymierne korzyści przy zasilaniu bateryjnym.

Optymalizacja programu wiąże się bezpośrednio ze zwiększeniem prędkości działania przy niezmienianej częstotliwości taktowania. Zastosowanie wydajniejszego mikrokontrolera z optymalną lista rozkazów może również zwiększyć szybkość wykonywania programu.



Rysunek 6. Rezygnacja z wywoływania funkcji z poziomu innych funkcji

Listing 9 Odczyt danych z ADC i obliczanie mediany próbek
```
void acquireData(void)
{
  uint32_t ch_index;
/* odblokowanie zegara HSI */
  RCC_HSICmd(ENABLE);
/* Odblokowanie zegara modułu ADC */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
/* Czekaj az HIS sie włączy */
  while (RCC_GetFlagStatus(RCC_FLAG_HSIRDY) == RESET)
  {}
/* odblokuj ADC */
  ADC_Cmd(ADC1, ENABLE);
/* czekaj az ADC sie właczy  (bedzie gotowy do działania) */
  while(ADC_GetFlagStatus(ADC1, ADC_FLAG_ADONS) == RESET);
/*16 pomiarów w pętli*/
  for (ch_index = 0; ch_index < MAX_CHNL; ch_index++)
  {
/* Programowy start konwersji ADC1 */
    ADC_SoftwareStartConv(ADC1);
/* Czekaj na zakończenie konwersji */
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET)
    { }
/* zapamiętanie wyniku */
    ADC_ConvertedValueBuff[ch_index] = ADC_GetConversionValue(ADC1);
  }
}
```

Zobaczyliśmy to na przykładzie zmiany listy rozkazów z 16-bitowej na 32-bitową. Jeżeli mamy do dyspozycji dużo pamięci programu Flash, to warto rezygnować z krótkich pętli oraz z wywoływania funkcji z poziomu innych funkcji, jak to zostało pokazane na **rysunku 6**.

Oczywistą metodą redukcji poboru energii jest racjonalne zarządzanie pracą układów peryferyjnych. Wszystkie nieużywane bloki peryferyjne powinny być wyłączone. Nieużywane linie GPIO najczęściej są ustawiane w trybie wejściowych linii analogowych. Eksperyment z wykorzystaniem kanału DMA pokazał, że opłaca się uśpić CPU powierzyć transfer danych z układów peryferyjnych do pamięci operacyjnej sterownikowi układów DMA.

Listing 10. Konfigurowanie kanału DMA
```
void configureDMA(void)
{
/* Declare NVIC init Structure */
  NVIC_InitTypeDef NVIC_InitStructure;
/* Enable DMA1 clock */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
/* De-initialise  DMA */
  DMA_DeInit(DMA1_Channel1);
/* DMA1 channel1 configuration */
  DMA_StructInit(&DMA_InitStructure);
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(ADC1->DR);          // Set DMA channel Peripheral
base address to ADC Data register
  DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC_ConvertedValueBuff;  // Set DMA channel Memeory base
addr to ADC_ConvertedValueBuff address
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;                         // Set DMA channel direction to
peripheral to memory
  DMA_InitStructure.DMA_BufferSize = ADC_CONV_BUFF_SIZE;                     // Set DMA channel buffersize to
peripheral to ADC_CONV_BUFF_SIZE
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;           // Disable DMA channel
Peripheral address auto increment
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;                    // Enable Memeory increment (To
be verified ....)
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;// set Peripheral data size to
8bit
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;              // set Memeory data size
to 8bit
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;                              // Set DMA in normal mode
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;                              // Set DMA channel
priority to High
  DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;                               // Disable memory to memory
option
  DMA_Init(DMA1_Channel1, &DMA_InitStructure);
// Use Init structure to initialise channel1 (channel linked to ADC)
/* Enable Transmit Complete Interrup for DMA channel 1 */
  DMA_ITConfig(DMA1_Channel1, DMA_IT_TC, ENABLE);
/* Setup NVIC for DMA channel 1 interrupt request */
  NVIC_InitStructure.NVIC_IRQChannel =   DMA1_Channel1_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

Listing 11 Konfiguracja DMA I ADC
```
void acquireDataDMA(void)
{
/* odblokowanie zegara  HSI */
  RCC_HSICmd(ENABLE);
/* czekaja az HIS wystartuje */
  while (RCC_GetFlagStatus(RCC_FLAG_HSIRDY) == RESET)
  {}
/* odblokowanie zegara ADC  */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
/* odblokowanie zegara DMA */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
/* odblokowanie modułu ADC1 */
  ADC_Cmd(ADC1, ENABLE);
/* Czekaja aż ADC1 będzie gotowy */
  while(ADC_GetFlagStatus(ADC1, ADC_FLAG_ADONS) == RESET);
/*reinicjalizacja licznika przesyłanych danych  DMA1_Channel1 data */
  DMA_SetCurrDataCounter(DMA1_Channel1, ADC_CONV_BUFF_SIZE);
/* odblokowanie kanału  DMA1_Channel1 */
  DMA_Cmd(DMA1_Channel1, ENABLE);
/* odblokowanie trybu  DMA dla  ADC1 */
  ADC_DMACmd(ADC1, ENABLE);
/* Start konwersji ADC */
  ADC_SoftwareStartConv(ADC1);
}
```

Jeżeli popatrzymy całościowo na pokazane tutaj przykłady to widać, że jest wiele sposobów na zaawansowane obniżanie poboru energii. To jak efektywnie będziemy potrafili je wykorzystać będzie zależało od wielu czynników. Po pierwsze, od rodzaju aplikacji. Dla układów, które da się sekwencyjnie i głęboko usypiać oszczędności mogą być bardzo duże, bo to tryby oszczędzania dają najbardziej wymierne korzyści. Kiedy już dobierzemy odpowiednie tryby oszczekania i efektywnie je zastosujemy, to dalsze oszczędności można uzyskać metodami prezentowanymi w tym artykule.

Tomasz Jabłoński, EP